SMP・マルチコアに対応した型付きアセンブリ言語

前田俊行　　米澤明憲

マルチコア環境等の、複数のプログラムが同時に実行されるような環境でも、静的型検査によりプログラムのメモリ安全性を保証・検証でき、ロック等の同期機構そのものを直接記述できるような型付きアセンブリ言語を提案する。

1　はじめに

OS カーネル等のシステムソフトウェアは従来、型安全なプログラミング言語で記述するのは難しいと考えられてきた。これは、メモリ管理機構やスレッド管理機構などを実現するために必要な、柔軟なメモリ操作やハードウェアに近い操作等を、従来の型安全なプログラミング言語で記述することができなかったためである。OS カーネルを安全な言語で記述する試みは幾つかなされてきたが [2] [7]、メモリ管理機構やスレッド管理機構などは対象としておらず、OS カーネルの本質的な部分以外を記述するのに留まっている。

これに対し我々は、型付きアセンブリ言語 [14] を用いた OS カーネルの記述手法について研究している。型付きアセンブリ言語とは、型検査によって型安全性が保証できるアセンブリ言語である。我々は、OS カーネルの記述が可能となるような型付きアセンブリ言語の型システムを設計し、実際に簡単な OS カーネル (メモリ管理機構とスレッド管理機構を含む) の構築も行った [10] [12] [18]。

しかし従来の型付きアセンブリ言語は、近年広く普及したマルチコア CPU のような、複数のプログラムが実際に同時に実行されるような環境には対応していなかった。

そこで我々は、このような SMP・マルチコア環境にも対応した型付きアセンブリ言語の設計を行う。特に我々は、OS カーネルなどのシステムソフトウェアを記述するために、スレッド管理機構やロック等の同期機構そのものを直接記述可能な型付きアセンブリ言語を設計する。

マルチコア環境に対応する上で問題となるのは、複数コア間で共有されるメモリ (以降、共有メモリと呼ぶ) の安全性とメモリコンシステンシであるが、本論文では、共有メモリの安全性について扱う。

以降の本論文の構成は次のとおりである。まず 2 節では、共有メモリの安全性を保証する手法を提案し、3 節では、提案する手法にもとづいた型付きアセンブリ言語を示す。次いで 4 節では関連研究について述べ、最後に 5 節でまとめと今後の課題について述べる。

2　共有メモリの安全性を保証する手法

2.1　共有メモリを扱う上での問題点

マルチコア環境など、複数のプログラムが同時に実行される環境では、一つのメモリ領域が同時に操作される可能性がある。このような環境でもメモリ安全性などを保証でき、かつ、システムソフトウェアを記述可能な型付きアセンブリ言語を実現するためには、「型を変更するようなメモリ操作 (strong update と呼ぶ)」「複数スレッド間でのポインタのエイリアス関

Toshiyuki Maeda and Akinori Yonezawa, 東京大学 大学院情報理工学系研究科, Graduate School of Computer Science and Technology, The University of Tokyo

係の追跡」の二つに対処する必要がある。

2.1.1 Strong update への対応

一般的な従来の型安全な言語上では、メモリ領域の型は不変であるので、仮にメモリ領域が同時に操作されても、メモリ安全性に関する限りは (その操作が atomic であるならば) 基本的に問題は生じない。

しかし、メモリ管理機構やスレッド管理機構などを記述するためには、メモリ領域の型をプログラムの実行中に変更すること (strong update) を許す必要がある。例えばメモリ管理機構の実装ではメモリ領域の確保や解放を行う必要があるが、このとき、ある型で確保された領域が、一度解放された後、別の型で再確保されるといったことが起きるため、メモリ領域の型の strong update は必要不可欠である。

この strong update を実現するために、シングルコア環境においては、我々は、エイリアス型 [17] と依存型の手法を統合した型システムを持つ型付きアセンブリ言語を設計・実装した [10]。具体的には、型システム上でポインタのエイリアス関係を静的に追跡することで、不正なメモリ操作を防いでいる。また実際に、メモリ安全性が保証されたメモリ管理機構やスレッド管理機構を実現した。

しかし、上述の型付きアセンブリ言語をマルチコア環境に対応させるには、次節で述べるもう一つの問題を考慮しなければならない。

2.1.2 複数スレッド間でのポインタのエイリアス関係の追跡

前節で述べた型付きアセンブリ言語をマルチコア環境に対応させるには、複数スレッド間でのポインタのエイリアス関係を追跡しなければならない。

従来の一般的なプログラミング言語では、同期機構が言語プリミティブとして用意されているため、この同期機構をヒントとしてエイリアス関係の解析を行うのが一般的である。例えば、プログラムがある共有メモリ領域を操作するときには、その領域に対応するロックが必ず取得されているかどうか等を解析すれば、複数のスレッドが同時に共有メモリを操作するか否かを判断することができる。

ところが我々の目的は、同期機構そのものを記述できるような型付きアセンブリ言語を設計することにあるため、一般的なプログラミング言語が提供するような同期機構はそもそも存在せず、利用できるのは CPU によって提供される同期命令 (例えば、同期スワップ命令、同期比較スワップ命令等) だけである。これらの同期命令を用いて、複数スレッド間のポインタのエイリアス関係を追跡しなければならない。

2.2 共有メモリの問題に対するアプローチ

前節で述べた共有メモリの問題に対して、我々は次のようなアプローチをとる。まず、共有メモリに対する strong update は、atomic なメモリ操作の間のみ許可するようにする。ただし、操作が終了した後の共有メモリの型が、操作開始時の型と等しくなるように型システムで制限する。ここで、atomic なメモリ操作とは、メモリ操作の結果が他の CPU に対して atomic に見える命令一つと、型に対する操作のみを行い、実行時には影響しない pseudo 命令の組み合わせとする。例えば、存在型を構築したり展開したりする pseudo 命令は、幾つ組み合わせても atomic なメモリ操作とみなす。

このアプローチにもとづくと、例えば、スピンロックは図 1 のように実装できる。まず、1 〜 4 行目と 13 〜 17 行目が、ラベル型を表す。ラベル型は、基本的にラベルごとに付与され、ラベルに実行が到達するときには、付与されたラベル型で示された条件が満たされることを型検査で検証する。例えば 1〜3 行目は、ラベル `lock` に実行が到達するときに満たされていなければならないメモリの状態を表し、4 行目がレジスタの状態を表している。より具体的には、1〜4 行目は、レジスタ `r1` が何らかの整数値 `p` を持ち (4 行目)、その整数値 `p` が示すアドレスには何らかのデータが存在していることを示している (1 行目) (今、このデータが複数のスレッド間で共有されているとする)。ここで、このデータを表している存在型は、基本的には二つの整数値からなるタプルであり、また、そのタプルのそれぞれの要素を整数値 `i`、`q` とすると、`i` が 0 であるときに限り、整数値 `q` が表すアドレスに何らかのデータ (ここでは型 `data`) が存在することを示している (2 行目と 3 行目)。より直観的にいえば、タプルの第一要素 (`i`) がロックを表し、

タプルの第二要素 (q) がそのロックで保護されているデータへのポインタを表している。

```
 1: { p -> exists(i).
 2:        { q -> data if [i == 0]}.
 3:        (i, q) }
 4: ( r1 : p )
 5: lock:
 6:    mov r2 <- 1
 7:    unpack r1
 8:    xchg [r1], r2
 9:    pack r1
10:    bne r2, 0, lock
11:    jmp unlock
12:
13: { p -> exists(i).
14:        { q -> data if [i == 0]}.
15:        (i, q),
16:   q -> data }
17: ( r1 : p )
18: unlock:
19:    unpack r1
20:    mov [r1] <- 0
21:    pack r1
22:    ...
```

図 1　同期ロックの表現方法

スピンロックの獲得 (lock) は以下のように行われる。まず、7 行目の unpack 命令により、アドレス p に存在するデータの存在型を展開する。すると、メモリの状態は以下のようになる。

```
{ p -> (i, q), q -> data if [i == 0] }
```

ここで、アドレス p の型が変更されているが、unpack 命令は実行時に影響のない命令なので型検査はパスする。次に 8 行目の xchg 命令により、レジスタ r2 が整数値 i を保持し、メモリの状態は以下のようになる。

```
{ p -> (1, q), q -> data if [i == 0] }
```

ここで、xchg 命令は atomic な命令のため、まだ操作開始前の型に戻す必要はない。次に 9 行目の pack 命令により、メモリの状態は以下のようになる。

```
{ p -> exists(i).
       { q -> data if [i == 0]}.
       (i, q)
  q -> data if [i == 0] }
```

(ここで、アドレス p の型が元に戻ったことに注意されたい。) ここまでの 7〜9 行目の命令が一つの atomic なメモリ操作を成している。

次に 10 行目の bne 命令により、整数値 i が 0 でないときにはラベル lock にジャンプし、0 であったときには引き続きラベル unlock にジャンプする。これは、ロックが既に確保されていた場合には、もう一度ロックの確保を試み、そうでない場合はロックの解放を行うことを表している (この例ではロックを確保した直後解放しているが、実際にはその間にアドレス q が指すデータの操作を行うことが可能である)。より具体的には、ラベル lock にジャンプするときには、$i \neq 0$ より、メモリの状態は以下のようになっているため、ラベル lock に付与されたラベル型の条件を満たしている。

```
{ p -> exists(i).
       { q -> data if [i == 0]}.
       (i, q) }
```

また、ラベル unlock にジャンプするときには、$i = 0$ より、メモリの状態は以下のようになっているため、ラベル unlock に付与されたラベル型の条件を満たしている。

```
{ p -> exists(i).
       { q -> data if [i == 0]}.
       (i, q)
  q -> data }
```

一方、スピンロックの解放 (unlock) では、まず、19 行目の unpack 命令により、メモリの状態は以下のようになる。

```
{ p -> (i, q), q -> data}
```

次に、20 行目の mov 命令により、メモリの状態は以下のようになる。

```
{ p -> (0, q), q -> data}
```

最後に、21 行目の pack 命令により、メモリの状態は以下のようになる。

```
{ p -> exists(i).
       { q -> data if [i == 0]}.
       (i, q) }
```

(ここで、アドレス p の型が元に戻ったことに注意されたい。) ここまでの 19〜21 行目の命令が一つの atomic なメモリ操作を成している。

以上のようにして、複数スレッドで同時に実行されてもメモリ安全性を損なわないスピンロックを表現することができる。

ここで試しに、図 1 の lock 中の xchg 命令を、三つの mov 命令で置き換えたものを考える (図 2)。この場合、atomic なメモリ操作は、7〜8 行目、9 行目、10〜11 行目の三つ存在するが、これらをまとめたものは atomic なメモリ操作とはならない。このため、9 行目の時点でメモリの状態は以下のようになるが、

```
{ p -> (i, q), q -> data if [i == 0] }
```

この時点で、型検査は失敗し、型エラーとなる。なぜなら、atomic なメモリ操作が終了した時点で、アドレス p に存在するデータの型が、開始時点の存在型に戻っていなければならないが、そうなっていないからである。実際、図 2 のプログラムは、8 行目と 9 行目の間で他のスレッドと競合状態となり得るため、スピンロックのロック獲得処理としては誤りである。

3 SMP・マルチコア環境に対応した型付きアセンブリ言語

本節では、前節で述べた手法にもとづいた型付きアセンブリ言語の設計の詳細について説明する。まず、抽象機械の構文を図 3 に、型の構文を図 4 に示す。(実際はより複雑であるが、ここでは簡略化したものを示している)。この型付きアセンブリ言語の型システムは、SMP・マルチコア環境においても、メモリ安全性が損なわれないことを型検査する。(より複雑な安全性、例えばデッドロックしないこと等は対象としない。)

```
 1: { p -> exists(i).
 2:        { q -> data if [i == 0]}.
 3:        (i, q) }
 4: ( r1 : p )
 5: lock:
 6:    mov r2 <- 1
 7:    unpack r1
 8:    mov r3 <- [r1]
 9:    mov [r1] <- r2
10:    mov r2 <- r3
11:    pack r1
12:    bne r2, 0, lock
13:    jmp unlock
14:    ...
```

図 2　誤った同期ロックの表現 (型エラーとなる)

$$
\begin{array}{lrcl}
(register) & r & ::= & r_1 \mid r_2 \mid \ldots \mid r_n \mid sp \\
(operand) & o & ::= & d \mid r \mid [r+d] \\
(inst.) & \iota & ::= & mov\ o \leftarrow o \mid bcc\ o \\
 & & & \mid jmp\ o \mid push\ o \mid pop\ o \\
 & & & \mid ret \mid cli \mid sti \\
 & & & \mid pushf \mid popf \mid iret \\
 & & & \mid pack_{[\bar{c}\mid\Psi]}\ o\ as\ \tau \\
 & & & \mid unpack\ o\ with\ \Delta \\
 & & & \mid block \mid unblock \\
(insts) & I & ::= & \cdot \mid \iota\ ; I \\
(tuple) & t & ::= & \langle d, \ldots, d \rangle \\
 & & & \mid pack_{[\bar{c}\mid\Psi]}.t \\
(value) & v & ::= & t \mid \forall\Delta.C.\Phi.i.i.I \\
(heap) & H & ::= & \cdot \mid \{d \mapsto v\}H \\
(registers) & R & ::= & \{r_1 \mapsto d, \ldots, r_n \mapsto d\} \\
(stack) & s & ::= & \cdot \mid d :: s \\
(processor) & P & ::= & (R, s, d_{pc}, d_{ipc}, d_{if}) \\
(state) & M & ::= & (H, \overline{P}, d_g)
\end{array}
$$

図 3　抽象機械の構文 (d は任意の整数値を表す)

従来の型付きアセンブリ言語と異なる点は、まず、抽象機械の状態 (M) が一つのメモリ (H) と、複数のプロセッサの状態 (P)、atomic フラグ (d_g) からなる点である (図 3)。抽象機械の各プロセッサの状態 (P) は、レジスタファイル (R)、スタック (s)、プログラムカウンタ (d_{pc})、割込みハンドラのアドレス (d_{ipc})、割込みフラグ (d_{if}) からなる。Atomic フラグは、具体的には、プロセッサが atomic なメモリ操作をしているかどうかをあらわす整数値であり、どのプロセッサも atomic なメモリ操作をしていない場合には 0、一つのプロセッサが atomic なメモリ操作をしている場合には、そのプロセッサのプロセッサ ID を保持する。

命令 *block* と命令 *unblock* が、抽象機械の atomic フラグを操作する命令である。直観的には、命令 *block* が atomic なメモリ操作の開始を、命令 *unblock* がその終了を表す。なお、このような命令は通常の CPU アーキテクチャには存在しないが、これらを導入した理由は、CPU アーキテクチャごとに異なる様々な同期命令を汎用的に扱うためである。

複数のメモリ操作命令と、*block*、*unblock* とを組み合わせることで、色々な種類の同期命令に対応することができる。例えば、IA-32 アーキテクチャ [8] の xchg 命令は以下のように表せる。

$block$; $mov\ r_{tmp} \leftarrow [r_d]$; $mov\ [r_d] \leftarrow r_s$;
$mov\ r_s \leftarrow r_{tmp}$; $unblock$

また、cmpxchg 命令は以下のように表せる。

(* cmpxchg *)
$block$; $mov\ r_{tmp} \leftarrow [r_d]$; $beq\ [r_d], r_c, cmpxchg\_eq$;
$mov\ r_c \leftarrow [r_d]$; $jmp\ cmpxchg\_end$
(* cmpxchg_eq *)
$mov\ [r_d] \leftarrow r_s$; $jmp\ cmpxchg\_end$
(* cmpxchg_end *)
$unblock$

割込みハンドラは、プロセッサの実行中に割込みが発生したときに実行される命令列である。より具体的には、割込みが発生すると、プロセッサは実行中のプログラムのアドレスと割込みフラグをスタックに保存し、割込みハンドラを実行する。

割込みフラグは、割込みの発生を制御するためのフラグである。具体的には、この割込みフラグを 0 にセットすると割込みは発生しない。0 以外の値にセットすると (たとえ割込みハンドラが実行されていたとしても) 割込みが (ランダムに) 発生し、割込みハンドラが実行される。

命令 *cli* と命令 *sti* はそれぞれ割込みフラグをクリアする命令とセットする命令である。また、命令 *pushf* はスタックに現在の割込みフラグの値を保存する命令である。逆に命令 *popf* はスタックに保存してある値を割込みフラグにセットする命令である。例えば、命令 *cli* を実行して割込みを禁止する前に、命令 *pushf* を実行して現在の割込みフラグを保存し、後で命令 *popf* を実行して保存した割込みフラグを復帰することで、割込み禁止処理を入れ子にすることもできる。

命令 *iret* は割込みハンドラから復帰するための命令である。具体的には、メモリスタックに保存されている帰りアドレスに、同じメモリスタックに保存されている値を割込みフラグにセットした状態でジャンプする。

残りの命令は従来の型付きアセンブリ言語と同様である。命令 *mov* はオペランド間でのデータの移送を行う。命令 *jmp* はオペランドが示す命令列へジャンプする。命令 *bcc* は、二つのオペランドが条件 *cc* を満たすとき、もう一つのオペランドが示す命令列へジャンプする。命令 *ret* は、スタックに積まれたアドレスが示す命令列へジャンプする。また、命令 *push* と命令 *pop* はスタックを操作する命令である。命令 *pack* と命令 *unpack* は、存在型を作成したり、展開したりする命令である。

なお、型そのものは、図 4 に示した通り、以前に我々が示した型付きアセンブリ言語とほぼ同様に、ヒープを表すヒープ型、レジスタを表すレジスタ型、スタックを表すスタック型からなる (ただし、ここでは簡単のため TALK の可変長配列型に関する部分は省略してある)。以前と異なる点は、命令列へのポインタを表すラベル型に、atomic フラグと割込みフラグを表す整数型 i が追加されている点 ($\forall\Delta.C.\Phi.i.i$) と、メモリの状態を表すヒープ型 (Ψ) の各要素に、

$$
\begin{array}{lrcl}
(\textit{type var}) & \alpha, \gamma, \epsilon, \rho & & \\
(\textit{type vars}) & \Delta & ::= & \cdot \mid \alpha, \Delta \mid \ldots \\
(\textit{int. type}) & i & ::= & \alpha \mid d \mid \ldots \\
(\textit{word type}) & \omega & ::= & \gamma \mid i \mid \forall \Delta.C.\Phi.i.i \\
(\textit{tuple type}) & \tau & ::= & \langle w, \ldots, w \rangle \mid \exists \Delta.C.\Psi.\tau \\
(\textit{heap type}) & \Psi & ::= & \cdot \mid \{i \mapsto \tau \textit{ if } C\}\ \Psi \\
 & & & \mid (\epsilon \textit{ if } C)\Psi \\
(\textit{stack type}) & \sigma & ::= & \rho \mid \cdot \mid w :: \sigma \\
(\textit{regs. type}) & \Gamma & ::= & \{r_1 \mapsto w; \ldots, r_n \mapsto w;\} \\
(\textit{store type}) & \Phi & ::= & (\Psi, \Gamma, \sigma) \\
(\textit{cop}) & cc & ::= & < \mid \leq \mid \ldots \\
(\textit{cstrts.}) & C & ::= & \cdot \mid i\ cc\ i, C
\end{array}
$$

図 4　型の構文

アクセス条件を表す C を指定できるようになっている点である。例えば、$\{i \mapsto \tau \textit{ if } i \neq 0\}$ というヒープ型は、アドレス i が示すメモリ領域には、$i \neq 0$ である場合に限り、型 τ を持つデータが保存されていることを表す。なお、特に条件を指定する必要がない場合は、$\{i \mapsto \tau\}$ のように省略して書くことにする。

3.1　操作的意味論

操作的意味論の定義は、図 5、図 6、図 7、図 8 のとおりである。なお、補助関数 *get* と *update* の定義は図 9 のとおりである。

まず、図 5 が、抽象機械全体の意味論を定義している。具体的には、atomic フラグが 0 であるときには、抽象機械中の任意のプロセッサが一段階実行される。0 でないときには、その数字に対応した ID を持つプロセッサを一段階実行する。つまりこれは、あるプロセッサが atomic なメモリ操作を行っているときには、他のプロセッサを実行しないことを表している。なお、メモリコンシステンシの観点からは、この定義は、メモリ操作がシーケンシャルコンシステンシ [1] を満たすことを意味している。近年の多くの CPU は、より制限の緩いコンシステンシを用いているが、本論文ではこの点に関しては扱わない。

各プロセッサの実行を表す操作的意味論は、基本的には従来の型付きアセンブリ言語を踏襲しているが、異なる点は割込みフラグの扱いである。具体的には、

$$
\begin{array}{c}
(H, (\ldots, P_i, \ldots), d_g) \mapsto_M (H', (\ldots, P_i', \ldots), d_g') \\
\text{where} \quad (H, P_i, d_g) \mapsto_{P,i} (H', P_i', d_g') \\
i = d_g \text{ if } d_g \neq 0 \\
\text{otherwise any processor id}
\end{array}
$$

図 5　抽象機械の操作的意味論: 抽象機械全体

atomic フラグ (d_g) が 0 であり、かつ、割込みフラグ (d_{if}) が 0 でないときにはいつでも割込みハンドラ (I_i) が実行される可能性がある (図 6)。なお、割込みハンドラが実行されるときには、割込みフラグが 0 にセットされる、すなわち、割込みが禁止される。さらに、割込みが発生したときの命令列 (I_p) へのポインタと割込みフラグがメモリスタックに保存される。

$$
\begin{array}{l}
(H, (R, s, d_{pc}, d_{ipc}, d_{if}), d_g) \ \mapsto_{P,pid}\ L' \\
\boxed{\begin{array}{ll}
L' = (H, P', d_g) & \\
\text{where} & P' = (R, s', d_{ipc}, d_{ipc}, 0) \\
 & s' = d_{pc} :: d_{if} :: s \\
 & (\text{if } d_g = 0 \wedge d_{if} \neq 0)
\end{array}}
\end{array}
$$

図 6　抽象機械の操作的意味論: 割込み処理

割込みが生じなかったときに実行される命令の操作的意味論は、図 7 と図 8 のとおりである。命令 *block* は、抽象機械の atomic フラグに自分が実行されたプロセッサの ID を保存する。逆に、命令 *unblock* は、atomic フラグを 0 にクリアする。命令 *cli* と命令 *sti* はそれぞれ割込みフラグに 0 、または 1 をセットする。命令 *pushf* は現在の割込みフラグの値をスタックに保存する。命令 *popf* は現在のスタックのトップに保存さている値をポップして割込みフラグにセットする。命令 *iret* は現在のスタックのトップに保存されている命令列へのポインタと、その次に保存されている割込みフラグの値をポップして、割込みフラグをポップした値にセットし、ポップしたポインタが指す命令列を実行する。その他の命令 (*mov*、*jmp*、*bcc*, *push*、*pop*, *ret*、*pack*、*unpack*) については従来の我々の型付きアセンブリ言語 [11] と同様である。

$(H,(R,s,d_{pc},d_{ipc},d_{if}),d_g) \mapsto_{P,pid} L'$

if $H[d_{pc}] =$	then $L' =$
$mov\ o_1 \leftarrow o_2$	$(H',(R',s',d'_{pc},d_{ipc},d_{if}),d_g)$ where $S=(H,R,s)$ $d=get(S,o_1)$ $(H',R',s')=update(S,o_2,d)$
$jmp\ o$	$(H,(R,s,d,d_{ipc},d_{if}),d_g)$ where $d=get((H,R,s),o)$
$bcc\ o_1,o_2,o_3$	$(H,(R,s,d,d_{ipc},d_{if}),d_g)$ where $S=(H,R,s)$ if $d_1\ cc\ d_2$ then $d=get(S,o_3)$ else $d=d'_{pc}$ $d_1=get(S,o_1)$ $d_2=get(S,o_2)$
$push\ o$	$(H,(R,s',d'_{pc},d_{ipc},d_{if}),d_g)$ where $S=(H,R,s)$ $s'=d::s \quad d=get(S,o)$
$pop\ o$	$(H',(R',s',d'_{pc},d_{ipc},d_{if}),d_g)$ where $S=(H,R,s) \quad s=d::s'$ $(H',R',\_)=update(S,o,d)$
ret	$(H,(R,s',d,d_{ipc},d_{if}),d_g)$ where $S=(H,R,s) \quad s=d::s'$

where $d'_{pc}=d_{pc}+1$

図 7　抽象機械の操作的意味論: 通常の命令 (1/2)

$(H,(R,s,d_{pc},d_{ipc},d_{if}),d_g) \mapsto_{P,pid} L'$

if $H[d_{pc}] =$	then $L' =$
cli	$(H,(R,s,d'_{pc},d_{ipc},0),d_g)$
sti	$(H,(R,s,d'_{pc},d_{ipc},1),d_g)$
$pushf$	$(H,(R,s',d'_{pc},d_{ipc},d_{if}),d_g)$ where $S=(H,R,s)$ $s'=d_{if}::s$
$popf$	$(H,(R,s',d'_{pc},d_{ipc},d'_{if}),d_g)$ where $S=(H,R,s)$ $s=d'_{if}::s'$
$iret$	$(H,(R,s',d,d_{ipc},d'_{if}),d_g)$ where $S=(H,R,s)$ $s=d::d'_{if}::s'$
$pack_{[\bar{c}\|\Psi]}\ o\ as\ \tau$	$(H',(R,s,d'_{pc},d_{ipc},d_{if}),d_g)$ where $H'=H\{d\mapsto t'\}$ $d=get((H,R,s),o)$ $t'=pack_{[\bar{c}\|\Psi]}.H(d)$
$unpack\ o\ with\ \Delta$	$(H',(R,s,d'_{pc},d_{ipc},d_{if}),d_g)$ where $H'=H\{d\mapsto t'\}$ $d=get((H,R,s),o)$ $H(d)=pack_{[\bar{c}\|\Psi]}.t'$
$block$	$(H,(R,s,d'_{pc},d_{ipc},d_{if}),pid)$
$unblock$	$(H,(R,s,d'_{pc},d_{ipc},d_{if}),0)$

where $d'_{pc}=d_{pc}+1$

図 8　抽象機械の操作的意味論: 通常の命令 (2/2)

$get((H,R,s),o) =$	
d	$(if\ o\ is\ d)$
$R(r)$	$(if\ o\ is\ r)$
$H(R(r)+c)$	$(if\ o\ is\ [r+c])$
$s[c]$	$(if\ o\ is\ [sp+c])$

$update((H,R,s),o,d) =$	
$H,R\{r\mapsto d\},s$	$(if\ o\ is\ r)$
$H\{(R(r)+c)\mapsto d\},R,s$	$(if\ o\ is\ [r+c])$
$H,R,s[c\mapsto d]$	$(if\ o\ is\ [sp+c])$

図 9　操作的意味論のための補助関数

3.2　型付け規則

図 10、図 11、図 13、図 14、図 15 は、我々の型付きアセンブリ言語の型付け規則の抜粋である。なお、*get_type* と *update_type* は、指定されたヒープやレジスタの型から、オペランドの型を取得する、または型を更新する補助関数である (図 12)。

我々の型付け規則の特徴は 2 つある。1 つは、atomic フラグや割込みフラグを型で追跡している点であり、もう 1 つは、複数プロセッサ間で共有されるメモリや、割込みハンドラとそれ以外のプログラム、もしくは割込みハンドラ間で共有されるメモリをヒープ型 Ψ_s、Ψ_b として特別に扱っている点である。Ψ_s が複

$$\frac{\vdash \Psi_s \qquad \vdash \Psi_b \qquad \vdash \Psi \quad \vdash H : \Psi \qquad \Psi \vdash_H P_i : \Gamma_i, \sigma_i \quad \not\models d_g \neq 0 \Rightarrow \vdash \Psi \to \Psi_s}{\vdash (H, \overline{P}, d_g) : (\Psi, \overline{(\Gamma, \sigma)})} \text{(STATE)}$$

$$\frac{\Psi \vdash R : \Gamma \qquad \Psi \vdash s : \sigma \quad \cdot, \cdot, (\Psi, \Gamma, \sigma), d_{if}, d_g \vdash H[d_{pc}] \quad \cdot, \cdot, (\Psi, \Gamma, \sigma), 0, 0 \vdash H[d_{ipc}] \quad \not\models d_{if} = 0 \Rightarrow \vdash \Psi \to \Psi_b}{\Psi \vdash_H (R, s, d_{pc}, d_{ipc}, d_{if}) : \Gamma, \sigma} \text{(PROCESSOR)}$$

$$\frac{\vdash \Psi \qquad \forall d \in Dom(H). \text{ if } H(d) = t \text{ then } \vdash t : \Psi(d) \text{ else if } H(d) = \forall \Delta.C.\Phi.i_i.i_g.I \text{ then } \Delta, C, \Phi, i_i, i_g \vdash I}{\vdash H : \Psi} \text{(HEAP)}$$

$$\frac{\vdash \Gamma \quad \forall r_i \in Dom(\Gamma). \Psi \vdash R(r_i) : \Gamma(r_i)}{\Psi \vdash R : \Gamma} \text{(REGISTER)}$$

$$\frac{\Psi \vdash d_i : w_i}{\Psi \vdash d_1 :: \ldots :: d_n : w_1 :: \ldots :: w_n} \text{(STACK)}$$

図 10　抽象機械の状態 (state) の型付け規則の抜粋

数プロセッサ間の共有メモリ、Ψ_b が割込みハンドラ等の間の共有メモリを表す。また、atomic フラグや各プロセッサの割込みフラグは、我々の型付け規則では単なる整数型として表わされる。

共有メモリの型 (Ψ_s、Ψ_b) を明示的に型システムで追跡することは、strong update (型を変更するようなメモリ操作) を実現するために必要である。Strong update はメモリ管理やスレッド管理を記述するために必須の機能である。

共有メモリの型追跡の基本的なアイデアは、2 節で述べたとおり、atomic なメモリ操作が実行されていないとき、また、割込みが発生するかもしれないときには strong update を禁止し、atomic なメモリ操作が実行されているとき、また、割込みが禁止されている場合には、一時的に strong update を許可するというものである。

例えば、規則 MOV は、atomic フラグが 0 でないと保証できないときには、命令 *mov* の実行に伴うメモリ操作後のヒープ型が、複数プロセッサ間での共有メモリの型 (Ψ_s) を含んでいなければならないことを示している ($\Delta, C \vdash \Phi' \to \Psi_s$)。また、割込みフラグが 0 であると保証できないときには、メモリ操作後のヒープ型が、割込みハンドラとの間での共有メモリのヒープ型 (Ψ_b) を含んでいなければならないことを示している ($\Delta, C \vdash \Phi' \to \Psi_b$)。

また、規則 UNBLOCK は、atomic フラグがクリアされる、すなわち、atomic なメモリ操作が終了して、他のプロセッサが共有メモリにアクセスする可能性があるので、命令 *unblock* 実行前のメモリのヒープ型が共有メモリのヒープ型 (Ψ_s) を含んでいなければならないことを示している。つまり、命令 *block* によって atomic なメモリ操作を開始した後では、共有メモリに対して strong update を行うことができるが、その後、命令 *unblock* によって操作を終了するときには、共有メモリの型は atomic なメモリ操作の開始前の型に戻っていなければならないということである。

以下、各規則について簡単に説明する。

規則 STATE は、抽象機械のメモリと各プロセッサが型付けできることを検査する。また、atomic フラグが 0 でないことが保証できないときは、複数プロセッサ間での共有メモリの型 (Ψ_s) をヒープ (の一部) が満たしていることも検査する。

規則 PROCESSOR は、レジスタファイル、スタックが型付けできること、また、プログラムカウンタ d_{pc} からはじまる命令列 ($H[d_{pc}]$)、また、割込みハンドラ ($H[d_{ipc}]$) が型付けできることを検査する。さらに、割込みフラグ d_{if} がクリアされ、割込みが禁止されていることが保証できないときは、割込みハンドラとの間での共有メモリの型 (Ψ_b) をヒープ (の一部) が満たしていることも検査する。

規則 HEAP は、単にヒープの全要素が、タプルとして型付け可能であるか、もしくは命令列として型付け可能であるかを検査する。また規則 REGISTER は、各レジスタが型付け可能であること、規則 STACK は、スタックの各要素が型付け可能であることを検査

する。

$$\frac{\begin{array}{c}\omega = \mathit{get\_type}(\Phi, o_2) \\ \Phi' = \mathit{update\_type}(\Phi, o_1, \omega) \\ \Delta, C \not\models i_i = 0 \Rightarrow \Delta, C \vdash \Phi' \rightarrow \Psi_b \\ \Delta, C \not\models i_g \neq 0 \Rightarrow \Delta, C \vdash \Phi' \rightarrow \Psi_s \\ \Delta, C, \Phi', i_i, i_g \vdash I\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash \mathit{mov}\ o_1, o_2;\ I}\ (\text{MOV})$$

$$\frac{\begin{array}{c}C'.\Phi'.i_i'.i_g' = \mathit{get\_type}(\Phi, o) \\ \Delta, C \vdash \Phi \leq \Phi' \\ \Delta, C \models C' \qquad \Delta, C \models i_i = i_i' \wedge i_g = i_g'\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash \mathit{jmp}\ o}\ (\text{JMP})$$

$$\frac{\begin{array}{c}i_1 = \mathit{get\_type}(\Phi, o_1) \\ i_2 = \mathit{get\_type}(\Phi, o_2) \\ C'.\Phi'.i_i'.i_g' = \mathit{get\_type}(\Phi, o_3) \\ \Delta, C \wedge (i_1\ \mathit{cc}\ i_2) \vdash \Phi \leq \Phi' \\ \Delta, C \wedge (i_1\ \mathit{cc}\ i_2) \models C' \\ \Delta, C \wedge (i_1\ \mathit{cc}\ i_2) \models i_i = i_i' \wedge i_g = i_t' \\ \Delta, C \wedge \neg(i_1\ \mathit{cc}\ i_2), \Phi, i_i, i_g \vdash I\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash \mathit{bcc}\ o_1, o_2, o_3;\ I}\ (\text{BCC})$$

$$\frac{\begin{array}{c}w = \mathit{get\_type}((\Psi, \Gamma, \sigma), o) \\ \Delta, C, (\Psi, \Gamma, w :: \sigma), i_i, i_g \vdash_{\Psi_s} I\end{array}}{\Delta, C, (\Psi, \Gamma, \sigma), i_i, i_g \vdash_{\Psi_s} \mathit{push}\ o;\ I}\ (\text{PUSH})$$

$$\frac{\begin{array}{c}\Phi' = \mathit{update\_type}((\Psi, \Gamma, \sigma), o, w) \\ \Delta, C \not\models i_i = 0 \Rightarrow \Delta, C \vdash \Phi' \rightarrow \Psi_b \\ \Delta, C \not\models i_g \neq 0 \Rightarrow \Delta, C \vdash \Phi' \rightarrow \Psi_s \\ \Delta, C, \Phi', i_i, i_g \vdash_{\Psi_s} I\end{array}}{\Delta, C, (\Psi, \Gamma, w :: \sigma), i_i, i_g \vdash_{\Psi_s} \mathit{pop}\ o;\ I}\ (\text{POP})$$

$$\frac{\begin{array}{c}\Delta, C \vdash (\Psi, \Gamma, \sigma) \leq \Phi \\ \Delta, C \models C' \qquad \Delta, C \models i_i = i_i' \wedge i_g = i_g'\end{array}}{\Delta, C, (\Psi, \Gamma, C'.\Phi.i_i'.i_g' :: \sigma), i_i, i_g \vdash_{\Psi_s} \mathit{ret}}\ (\text{RET})$$

図 11　命令の型付け規則

規則 MOV は、オペランド o_1 のあらわす型をオペランド o_2 の型で更新し、残りの命令列 I を型付けする。また、前述のとおり、atomic フラグを表す型 i_g が 0 でないことを保証できないときには、ヒープ型が複数プロセッサ間の共有メモリのヒープ型 Ψ_s を含んでいることを検査し、割込みフラグを表す型 i_i が 0 であることを保証できないときには、ヒープ型が共有メモリのヒープ型 Ψ_b を含んでいることを検査する。これにより、もしこの *mov* 命令の直後に他のプロセッサが共有メモリにアクセスしたり、割込みが発生したとしても、共有メモリの型が保存されていることを保証できる。

規則 JMP は、オペランド o の型がラベル型であることを確認し、現在のストアの型 Φ が、このラベル

$\mathit{get\_type}(\Delta, C, (\Psi, \Gamma, \sigma), o) =$	
d	$(\mathit{if}\ o\ \mathit{is}\ d)$
$\Gamma(r)$	$(\mathit{if}\ o\ \mathit{is}\ r)$
w_d	where $\Psi = \{i \mapsto \langle \ldots, w_d, \ldots \rangle\ \mathit{if}\ C'\}\Psi'$ $\Delta, C \models i = \Gamma(r)$ $\Delta, C \models C'$ $(\mathit{if}\ o\ \mathit{is}\ [r + d])$
w_d	where $\sigma = \ldots :: w_d :: \sigma'$ $(\mathit{if}\ o\ \mathit{is}\ [sp + d])$

$\mathit{update\_type}(\Delta, C, (\Psi, \Gamma, \sigma), o, w) =$	
(Ψ, Γ', σ)	where $\Gamma' = \Gamma\{r \mapsto w\}$ $(\mathit{if}\ o\ \mathit{is}\ r)$
(Ψ'', Γ, σ)	where $\Psi = \{i \mapsto \langle \ldots, w_d, \ldots \rangle\ \mathit{if}\ C'\}\Psi'$ $\Delta, C \models i = \Gamma(r)$ $\Delta, C \models C'$ $\Psi'' = \{i \mapsto \langle \ldots, w, \ldots \rangle\ \mathit{if}\ C'\}\Psi'$ $(\mathit{if}\ o\ \mathit{is}\ [r + d])$
(Ψ, Γ, σ')	where $\sigma = \ldots :: w_d :: \sigma'$ $\sigma' = \ldots :: w :: \sigma'$ $(\mathit{if}\ o\ \mathit{is}\ [sp + d])$

図 12　型付け規則のための補助関数

型で指定されているストアの制約 Φ' を満たしていることを検査する。また、ラベル型で指定されている制約 C' を現在の制約 C から導出できるかも検査し、更にラベル型で指定されている atomic 操作の状態や割込み状態を現在の atomic フラグと割込みフラグが満たしているかも検査する ($\Delta, C \models i_i = i_i' \wedge i_g = i_g'$)。

規則 BCC は、まず、オペランド o_1 と o_2 が整数型を持つこと、o_3 の型がラベル型であることを確認する。次に、分岐の条件が満たされることを仮定して ($C \wedge i_1\ cc\ i_2$)、現在のストアの型 Φ が、o_3 のラベル型で指定されているストアの制約 Φ' を満たしていることを検査する。また、そのラベル型で指定されている制約 C' を導出できるかも検査し、更にラベル型で指定されている atomic 操作の状態や割込み状態を現在の atomic フラグと割込みフラグが満たしているかも検査する。最後に、分岐の条件が満たされていないことを仮定して ($C \wedge \neg(i_1\ cc\ i_2)$)、残りの命令列 I を型付けする。

規則 PUSH は、オペランド o の型を現在のスタック型 σ に連結し、この新しいスタック型で残りの命令の型付けを行う。規則 MOV とは異なり、atomic フラグや割込みフラグと共有メモリに関する検査は不要である。これは、共有メモリにはスタックは含まれず、従って命令 *push* によってヒープ型が変化することはないからである。

規則 POP は、現在のスタックの型 $w :: \sigma$ の先頭からワード型 w を削除する。またこのワード型 w を用いてストア型中のオペランド o に対応した部分を更新し、残りの命令列 I の型付けを行う。規則 $PUSH$ とは異なり、命令 *pop* はヒープ型を更新する可能性があるため、現在の atomic フラグ i_g が 0 でないことを保証できない場合は、現在のヒープ (の一部) が複数プロセッサ間の共有メモリの型 (Ψ_s) を満たすことを、また割込みフラグ i_i が 0 であることを保証できない場合には、割込みハンドラとの間の共有メモリの型 (Ψ_b) を満たすことを検査する。

規則 RET は、現在のスタック型の先頭のワード型を削除し、そのワード型がラベル型であり、更にそのラベル型で指定された制約が、現在のストア型、制約、割込みフラグによって満たされているかを検査する。命令 *push* 同様、命令 *ret* はヒープ型を更新しないので、共有メモリの型に関する検査は不要である。

$$\frac{\Delta, C, \Phi, i_i, 1 \vdash I}{\Delta, C, \Phi, i_i, i_g \vdash block;\ I}\ (\textsc{block})$$

$$\frac{\Delta, C \vdash \Phi \rightarrow \Psi_s \qquad \Delta, C, \Phi, i_i, 0 \vdash I}{\Delta, C, \Phi, i_i, i_g \vdash unblock;\ I}\ (\textsc{unblock})$$

図 13　メモリの atomic 操作に関する命令の型付け規則

規則 BLOCK は、atomic フラグを 1 にセットして残りの命令の型付けを行う。命令 *block* はオペランドを取らず、またスタック、ヒープ、レジスタの更新も行わないため、型付けは必ず成功する。

規則 UNBLOCK は、atomic フラグを 0 にクリアして残りの命令の型付けを行う。命令 *block* 同様、命令 *unblock* もスタック、ヒープ、レジスタの更新は行わないが、命令実行の直後に、他のプロセッサが共有メモリを操作する可能性があるため、現在のストア型が、複数プロセッサ間での共有メモリの型 Ψ_s を満たしているかを検査する必要がある。

規則 CLI は、割込みフラグを 0 にして残りの命令の型付けを行う。命令 *cli* はオペランドを取らず、またスタック、ヒープ、レジスタの更新も行わないため、型付けは必ず成功する。

規則 STI は、割込みフラグを 1 にして残りの命令の型付けを行う。命令 *cli* 同様、命令 *sti* もスタック、ヒープ、レジスタの更新は行わないが、命令実行の直後で割込みが発生する可能性があるため、現在のストア型が、割込みハンドラが想定する共有メモリの型 Ψ_b を満たしているかを検査する。

規則 PUSHF は、現在の割込みフラグ i_i をスタックに連結して残りの命令の型付けを行う。命令 *pushf* が更新するのはスタックのみであり、またスタックは共有メモリ型 Ψ_b には含まれていないので、規則 PUSH 同様、割込みフラグと共有メモリ型 Ψ_b に関する検査は必要ない。

規則 POPF は、現在のスタック型の先頭のワード型を削除し、これを割込みフラグにセットして残りの命令の型付けを行う。命令 *pop* とは異なり命令 *popf*

$$\frac{\Delta, C, \Phi, 0, i_g \vdash I}{\Delta, C, \Phi, i_i, i_g \vdash cli; I} \text{(CLI)}$$

$$\frac{\Delta, C \vdash \Phi \to \Psi_b \qquad \Delta, C, \Phi, 1, i_g \vdash I}{\Delta, C, \Phi, i_i, i_g \vdash sti; I} \text{(STI)}$$

$$\frac{\Delta, C, (\Psi, \Gamma, i_i :: \sigma), i_i, i_g \vdash I}{\Delta, C, (\Psi, \Gamma, \sigma), i_i, i_g \vdash pushf;\ I} \text{(PUSHF)}$$

$$\frac{\begin{array}{c}\Delta, C, (\Psi, \Gamma, \sigma), i', i_g \vdash I \\ \Delta, C \not\models i' = 0 \Rightarrow \Delta, C \vdash \Psi \supseteq \Psi_b\end{array}}{\Delta, C, (\Psi, \Gamma, i' :: \sigma), i_i, i_g \vdash_{\Psi_s} popf;\ I} \text{(POPF)}$$

$$\frac{\begin{array}{c}\Delta, C \vdash (\Psi, \Gamma, \sigma) \leq \Phi \\ \Delta, C \not\models i_1 = 0 \Rightarrow \Delta, C \vdash \Psi \supseteq \Psi_b \\ \Delta, C \models C' \qquad \Delta, C \models i_1 = i_2 \wedge i_g = i'_g\end{array}}{\begin{array}{c}\Delta, C, (\Psi, \Gamma, (C', \Phi, i_2, i'_g) :: i_1 :: \sigma), i_i, i_g \\ \vdash iret\end{array}} \text{(IRET)}$$

図 14　割込みに関する命令の型付け規則

はヒープを更新することはないが、割込みフラグが更新されるため、割込みフラグが 0 であることを確認できない場合には現在のヒープ型 Ψ が、割込みハンドラとの間の共有メモリのヒープ型 Ψ_b を包含していることを検査する。

規則 IRET は、基本的に規則 RET と規則 POPF の組み合わせと同等である。

規則 PACK は、まずオペランド o の示す型が、命令 *pack* に指定された存在型の型変数 (Δ') を、指定された型 ($\bar{c}$) で置換したものと等しいことを検査する。また、命令に指定されたヒープ型 Ψ_1 が現在のヒープ型に含まれおり、かつ、指定された存在型に示されたヒープ型の条件を満たしているかを検査する。更に、存在型に指定された制約 C' が満たされていることを検査する。最後に、現在のヒープ型から Ψ_1 を除き、オペランド o の示す型を指定された存在型で更新して、残りの命令列の型付けを行う。

規則 UNPACK は、まずオペランド o の示す型が存在型であることを確認する。次に、その存在型に示された制約やヒープ型、タプル型を展開し、それらで現在の制約やヒープ型を更新して、残りの命令列の型付けを行う。

$$\frac{\begin{array}{c}\Phi \equiv (\Psi, \Gamma, \sigma) \qquad \tau \equiv \exists \Delta'.C'.\Psi'.\tau' \\ \Delta, C \vdash \Psi = \Psi_1\Psi_2 \quad \Delta, C \vdash \Psi_1 = \Psi'[\bar{c}/\Delta'] \\ get\_type((\Psi_2, \Gamma, \sigma), o) = \tau'[\bar{c}/\Delta'] \\ \Phi' = update\_type((\Psi_2, \Gamma, \sigma), o, \tau) \\ \Delta, C \models C'[\bar{c}/\Delta'] \qquad \Delta, C, \Phi', i_i, i_g \vdash I\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash pack_{[\bar{c}|\Psi_1]}\ o\ as\ \tau; I} \text{(PACK)}$$

$$\frac{\begin{array}{c}get\_type(\Phi, o) = \exists \Delta'.C'.\Psi'.\tau' \\ (\Psi_1, \Gamma', \sigma') = update\_type(\Phi, o, \tau'') \\ C'' \equiv C'[\Delta''/\Delta'] \qquad \Psi_2 \equiv \Psi'[\Delta''/\Delta'] \\ \tau'' \equiv \tau'[\Delta''/\Delta'] \\ \Delta\Delta'', C \wedge C'', (\Psi_1\Psi_2, \Gamma', \sigma') \vdash I\end{array}}{\Delta, C, \Phi, i_i, i_g \vdash unpack\ o\ with\ \Delta''; I} \text{(UNPACK)}$$

図 15　存在型の操作に関する命令の型付け規則

4　関連研究

高級言語のレベルにおいて、複数スレッドが同時に実行される環境における同期機構などを扱う研究は多く存在するが [4] [9] [6] [5]、これらの研究は競合状態の防止などが主目的であり、ロックなどの同期機構がプリミティブとして仮定されているため、システムソフトウェアに必要な同期機構そのものの記述に直接応用することはできない。これに対し本論文は、アセンブリ言語のような機械語に近い言語において、CPU アーキテクチャが提供する同期命令のみを用いて同期機構そのものを記述可能な型システムを提案した。

Vasconcelos らは、マルチコア環境におけるロックに対応した型付きアセンブリ言語を提案した [16]。彼らの型付きアセンブリ言語では、ロックやロックに対する操作 (lock や unlock) がプリミティブとして用意されているため、同期機構そのものを記述したり、そのメモリ安全性などを保証することはできない。これに対し、我々の型付きアセンブリ言語は、ロックやロックに対する操作が特別なプリミティブとして存在せず、CPU アーキテクチャが提供する同期命令のみに依存するため、2 節で見たように同期機構そのものの記述が可能である。また、彼らの型付けアセンブリ言語では、スレッドの複製 (fork) もプリミティブとして提供されているが、我々の型付きアセンブリ言語

では、(我々の先行研究 [10][18] と組み合わせることで) スレッド管理機構そのものの記述も可能である。

我々は [18] において、ハードウェア割込みに対応した型付きアセンブリ言語を提案した。具体的には、通常のプログラム実行と、割込み処理ハンドラの間で共有されるメモリの型安全性を、割込み禁止中にのみ共有メモリの strong update を許可し、割込みが再許可されたときに共有メモリの型を元に戻すことで保証した。ただし、実際に複数のプログラムが同時に実行される SMP・マルチコア環境には対応していなかった。また Feng らは、通常のプログラム実行と割込み処理ハンドラ間の共有メモリの性質を、[18] と同様の方針で手動で検証する手法を提案した [3]。しかし、彼らの手法も SMP 環境等には対応していない。一方、我々の手法がシンプルなメモリ安全性のみを対象にしているのに対し、彼らの手法は、separation logic [15] にもとづいたより汎用的な検証を行うため、より広い範囲の性質を検証することができる。

5 まとめと今後の課題

本論文は、SMP やマルチコアなどの複数のプログラムが同時に実行されるような環境において、プログラムの型安全性を型検査によって保証・検証でき、ロック等の同期機構そのものを記述可能な型付きアセンブリ言語を提案した。

今後の課題は二つある。一つは、本論文で扱わなかった、シーケンシャルコンシステンシ以外のメモリコンシステンシ [1] に対応することである。もう一つは、readers-writer ロックや read-copy-update [13] などの、より複雑な同期機構を記述できるように型システムを更に拡張することである。

参考文献

[1] Adve, S. V. and Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial, *IEEE Computer*, Vol. 29,No. 12(1996), pp. 66–76.

[2] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C.: Extensibility, Safety and Performance in theh SPIN Operating System, *15th Symposium on Operating Systems Principles*, 1995, pp. 267–284.

[3] Feng, X., Shao, Z., Dong, Y., and Guo, Y.: Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008, pp. 170–182.

[4] Flanagan, C. and Abadi, M.: Object Types against Races, *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, 1999, pp. 288–303.

[5] Flanagan, C. and Freund, S. N.: Type inference against races, *Sci. Comput. Program.*, Vol. 64,No. 1(2007), pp. 140–165.

[6] Grossman, D.: Type-safe multithreading in cyclone, *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, 2003, pp. 13–25.

[7] Hunt, G. C., Larus, J. R., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., and Zill, T. W. B.: An Overview of the Singularity Project, Technical Report MSR-TR-2005-135, Microsoft Corporation, 2005.

[8] Intel Corporation: IA-32 Intel Architecture. http://developer.intel.com.

[9] Iwama, F. and Kobayashi, N.: A new type system for JVM lock primitives, *ASIA-PEPM*, 2002, pp. 71–82.

[10] Maeda, T. and Yonezawa, A.: Writing practical memory management code with a strictly typed assembly language, *Third workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2006.

[11] Maeda, T.: Typed Assembly Language for Kernel. http://web.yl.is.s.u-tokyo.ac.jp/~tosh/talk/.

[12] Maeda, T.: *Writing an Operating System with a Strictly Typed Assembly Language*, PhD Thesis, University of Tokyo, 2006.

[13] McKenney, P. E. and Slingwine, J. D.: Read-Copy Update: Using Execution History to Solve Concurrency Problems, *Parallel and Distributed Computing and Systems*, 1998, pp. 509–518.

[14] Morrisett, G., Walker, D., Crary, K., and Glew, N.: From System F to Typed Assembly Language, *ACM Transactions on Programming Languages and Systems*, Vol. 21,No. 3(1999), pp. 528–569.

[15] Reynolds, J. C.: Separation Logic: A Logic for Shared Mutable Data Structures, *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[16] Vasconcelos, V. T. and Martins, F.: A Multithreaded Typed Assembly Language, *Proceedings of TV06 - Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, 2006, pp. 133–141.

[17] Walker, D. and Morrisett, G.: Alias Types for Recursive Data Structures, *Types in Compilation*, 2000.

[18] 前田俊行, 米澤明憲: 割込みに対応した型付きアセンブリ言語, **第 *5* 回ディペンダブルシステムワークショップ** *(DSW07)*, 2007.